brief contents

PART 1 THE LANGUAGE 1

1 ■ First steps 3
2 ■ Building blocks 18
3 ■ Control flow 63
4 ■ Data abstractions 101

PART 2 THE PLATFORM 131

5 ■ Concurrency primitives 133
6 ■ Generic server processes 164
7 ■ Building a concurrent system 181
8 ■ Fault-tolerance basics 201
9 ■ Isolating error effects 222
10 ■ Sharing state 247

PART 3 PRODUCTION 263

11 ■ Working with components 265
12 ■ Building a distributed system 290
13 ■ Running the system 321

9 Isolating error effects

This chapter covers

- Understanding supervision trees
- Starting workers dynamically
- "Let it crash"

In chapter 8, you learned about the basic theory behind error handling in concurrent systems based on the concept of supervisors. The idea is to have a process whose only job is to supervise other processes and to restart them if they crash. This gives you a way to deal with all sorts of unexpected errors in your system. Regardless of what goes wrong in a worker process, you can be sure that the supervisor will detect an error and restart the worker.

In addition to providing basic error detection and recovery, supervisors play an important role in isolating error effects. By placing individual workers directly under a supervisor, you can confine an error's impact to a single worker. This has an important benefit: it makes your system more available to its clients. Unexpected errors will occur no matter how hard you try to avoid them. Isolating the effects of such errors allows other parts of the system to run and provide service while you're recovering from the error.

For example, a database error in this book's example to-do system shouldn't stop the cache from working. While you're trying to recover from whatever went wrong in the database part, you should continue to serve existing cached data, thus providing at least partial service. Going even further, an error in an individual database worker shouldn't affect other database operations. Ultimately, if you can confine an error's impact to a small part of the system, your system can provide most of its service all of the time.

Isolating errors and minimizing their negative effects is the topic of this chapter. The main idea is to run each worker under a supervisor, which makes it possible to restart each worker individually. You'll see how this works in the next section, where you start to build a fine-grained supervision tree.

9.1 Supervision trees

In this section, we'll discuss how to reduce the effect of an error on the entire system. The basic tools are processes, links, and supervisors, and the general approach is fairly simple. You always have to consider what will happen to the rest of the system if a process crashes due to an error; and you should take corrective measures when an error's impact is too wide (the error affects too many processes).

9.1.1 Separating loosely dependent parts

Let's look at how errors are propagated in the to-do system. Links between processes are depicted in figure 9.1.

As you can see in the diagram, the entire structure is connected. Regardless of which process crashes, the exit signal will be propagated to its linked processes. Ultimately, the to-do cache process will crash as well, and this will be noticed by the `Todo.Supervisor`, which will in turn restart the cache process.

This is a correct error-handling approach because you restart the system and don't leave behind any dangling processes. But such a recovery approach is too coarse.

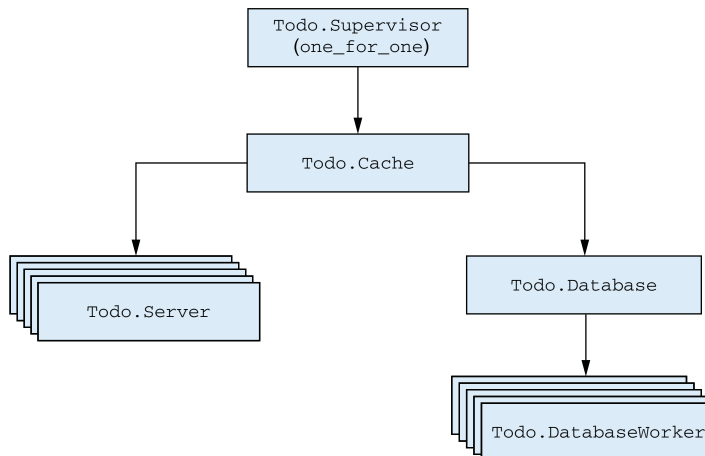

Figure 9.1 Process links in the to-do system

Wherever an error happens, the entire system is restarted. In the case of a database error, the entire to-do cache will terminate. Similarly, an error in one to-do server process will take down all the database workers.

This coarse-grained error recovery is due to the fact that you're starting worker processes from within other workers. For example, a database server is started from the to-do cache. To reduce error effects, you need to start individual workers from the supervisor. Such a scheme makes it possible for the supervisor to supervise and restart each worker separately.

Let's see how to do this. First, you'll move the database server so it's started directly from the supervisor. This will allow you to isolate database errors from those that happen in the cache.

Placing the database server under supervision is simple enough. You must remove the call to `Todo.Database.start_link` from `Todo.Cache.init/1`. Then, you have to add another child to the supervisor specification, as illustrated in the following listing.

Listing 9.1 Supervising database server (supervise_database/lib/todo/supervisor.ex)

```
defmodule Todo.Supervisor do
  ...

  def init(_) do
    processes = [
      worker(Todo.Database, ["./persist/"]),     <-- Starts the database server
      worker(Todo.Cache, [])
    ]
    supervise(processes, strategy: :one_for_one)
  end
end
```

These changes ensure that the cache and the database are separated, as shown in figure 9.2. Running both the database and cache processes under the supervisor makes it possible to restart each worker individually. An error in the database worker will crash the entire database structure, but the cache will remain undisturbed. This means all clients reading from the cache will be able to get their results while the database part is restarting.

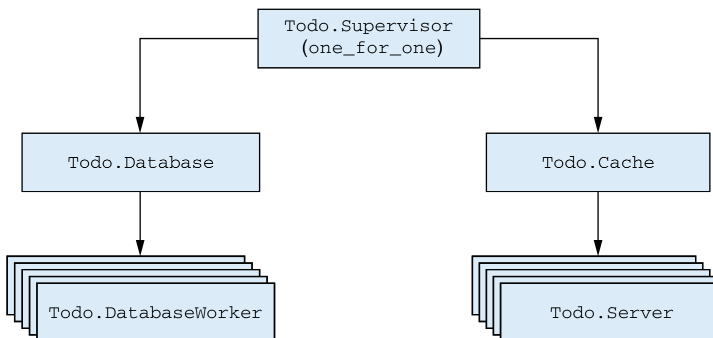

Figure 9.2 Separated supervision of the database and the cache

Let's verify this. Go to the supervise_database folder, and start the shell (`iex -S mix`). Then start the system:

```
iex(1)> Todo.Supervisor.start_link
Starting database server.
Starting database worker.
Starting database worker.
Starting database worker.
Starting to-do cache.
```

Now, kill the database server:

```
iex(2)> Process.whereis(:database_server) |> Process.exit(:kill)
Starting database server.
Starting database worker.
Starting database worker.
Starting database worker.
```

As you can see from the output, only database-related processes are restarted. The same is true if you terminate the to-do cache. By placing both processes under supervision, you localize the negative impact of an error. A cache error will have no effect on the database part, and vice versa.

Recall chapter 8's discussion of process isolation. Because each part is implemented in a separate process, the database server and the to-do cache are isolated and don't affect each other. Of course, these processes are indirectly linked via the supervisor, but the supervisor is trapping exit signals, thus preventing further propagation. This is in particular a property of `one_for_one` supervisors—they confine an error's impact to a single worker and take the corrective measure (restart) only on that process.

Child processes are started synchronously

In this example, the supervisor starts two child processes. It's important to be aware that children are started synchronously, in the order specified. The supervisor starts a child, waits for it to finish, and then moves on to start the next child. When the worker is a `gen_server`, the next child is started only after the `init/1` callback function for the current child is finished.

You may recall from chapter 7 that `init/1` shouldn't run for a long time. This is precisely why. If `Todo.Database` was taking, say, five minutes to start, you wouldn't have the to-do cache available all that time. Always make sure your `init/1` functions run fast, and use the trick mentioned in chapter 7 (a process that sends itself a message during initialization) when you need more complex initialization.

9.1.2 Rich process discovery

Although you have some basic error isolation, there's still a lot to be desired. An error in one database worker will crash the entire database structure and terminate all running database operations. Ideally, you want to confine a database error to a single worker. This means each database worker has to be directly supervised.

There is one problem with this approach. Recall that in the current version, the database server starts the workers and keeps their pids in its internal list. But if a process is started from a supervisor, you don't have access to the pid of the worker process. This is a property of the Supervisor pattern. You can't keep a process's pid for a long time, because that process might be restarted, and its successor will have a different pid.

Therefore, you need a way to give symbolic names to supervised processes and access each process via this name. When a process is restarted, the successor will register itself under the same name, which will allow you to reach the right process even after multiple restarts.

You could use registered aliases for this purpose. The problem is that aliases can only be atoms, and in this case you need something more elaborate that will allow you to use arbitrary terms, such as `{:database_worker, 1}`, `{:database_worker, 2}`, and so on. What you need is a process registry that maintains a key-value map, where the keys are aliases and the values are pids. A process registry differs from standard local registration in that aliases can be arbitrarily complex.

Every time a process is created, it can register itself to the registry under an alias. If a process is terminated and restarted, the new process will re-register itself. So, having a registry will give you a fixed point where you can discover processes (their pids). Because you need to maintain a long-running state, the registry needs to be a process. The idea is illustrated in figure 9.3.

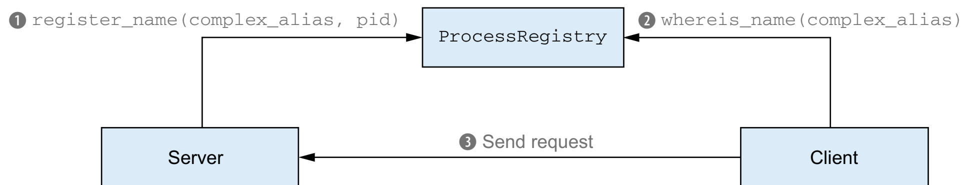

Figure 9.3 Discovering processes through a registry

In step 1, the worker process registers itself during initialization. Some time later, the client process (`Todo.Server`) will query the registry for the pid of the desired worker. The client can then issue a request to the server process. Let's see a sample usage of the registry:

```
iex(1)> Todo.ProcessRegistry.start_link

iex(2)> Todo.ProcessRegistry.register_name(
          {:database_worker, 1},
          self
        )
:yes

iex(3)> Todo.ProcessRegistry.whereis_name({:database_worker, 1})
#PID<0.55.0>
```

The process registers itself.

Retrieves a process ID

This interface seems somewhat strange, with both functions having elaborate names and `register_name` returning `:yes` in the case of a successful registration (or `:no` if a process under the given alias is already registered). There is a special reason I chose this interface. By implementing exactly these kind of functions, you can combine your registry with the `gen_server` behaviour.

You'll see how this works in a minute. First you need to implement two more functions `unregister_name/1`, which unregisters the process, and `send/2`, which sends a message to the process identified via alias. Let's see how these functions can be used:

```
iex(4)> Todo.ProcessRegistry.send({:database_worker, 1}, "Hello")    # Sends a message via registered alias
iex(5)> receive do msg -> msg end
"Hello"

iex(6)> Todo.ProcessRegistry.unregister_name({:database_worker, 1})    # Unregisters the process
iex(7)> Todo.ProcessRegistry.whereis_name({:database_worker, 1})
:undefined
```

Notice that `Todo.ProcessRegistry` is itself a `gen_server` registered under a local alias. How can you tell? Because the process obviously maintains server-wide state (mapping of alias to pid), and none of its functions require the pid of the process registry.

Once you have a module with this interface, you can combine it with `GenServer` using *via tuples*. A via tuple must be in the form `{:via, module, alias}`. You can send such tuples to `GenServer` functions such as `start_link`, `call`, and `cast`; `GenServer` will in turn use functions from the given module to register a process, discover it, and send messages to it. Take a look at the following examples:

```
GenServer.start_link(
  Todo.DatabaseWorker,
  db_folder,
  name: {:via, Todo.ProcessRegistry, {:database_worker, 1}}    # Uses Todo.ProcessRegistry for alias registration
)

GenServer.call(
  {:via, Todo.ProcessRegistry, {:database_worker}, 1},    # Uses Todo.ProcessRegistry for pid discovery
  {:get, key}
)

GenServer.cast(
  {:via, Todo.ProcessRegistry, {:database_worker}, 1},    # Uses Todo.ProcessRegistry for pid discovery
  {:store, key, data}
)
```

The implementation of the process registry is mostly a rehash of previously presented techniques. The registry is a simple `gen_server` that maintains the alias-to-pid mapping as a `HashDict`. Process discovery then amounts to querying the `HashDict` instance. The relevant `GenServer` callbacks are presented in the following listing.

Listing 9.2 Registering a process (pool_supervision/lib/todo/process_registry.ex)

```
defmodule Todo.ProcessRegistry do
  use GenServer

  ...

  def init(_) do
    {:ok, HashDict.new}
  end

  def handle_call({:register_name, key, pid}, _, process_registry) do
    case HashDict.get(process_registry, key) do
      nil ->
        Process.monitor(pid)                                    # Monitors the registered process
        {:reply, :yes, HashDict.put(process_registry, key, pid)}  # Stores the alias-to-pid mapping
      _ ->
        {:reply, :no, process_registry}
    end
  end

  def handle_call({:whereis_name, key}, _, process_registry) do
    {
      :reply,
      HashDict.get(process_registry, key, :undefined),    # Retrieves the pid for the given alias
      process_registry
    }
  end

  ...
end
```

Notice how, during the registration, you also set up a monitor to the registered pid. This will allow you to detect the termination of a registered process and react to it by removing corresponding entries from the registry state. This is important because it lets you re-register the restarted process under the same name.

Why are you using monitors instead of links? Links aren't appropriate in this situation because a crashed registered process would take down the registry and, by extension, all other registered processes. What you need here is unidirectional crash propagation. The registry should be notified about the termination of registered processes, but not vice versa. This is what monitors are useful for.

Of course, you need to handle the `:DOWN` message, which informs you that a registered process has terminated:

```
defmodule Todo.ProcessRegistry do
  ...

  def handle_info({:DOWN, _, :process, pid, _}, process_registry) do
    {:noreply, deregister_pid(new_registry, pid)}
  end

  ...
end
```

Here you invoke the private `deregister_pid/2` function, which removes all corresponding entries from the process registry state. The implementation of this function is straightforward, so it isn't presented here.

One thing remains. To be able to work with `GenServer` and via tuples, the `Todo.ProcessRegistry` module must implement and export the `send/2` function, which must send the message to the process identified by the provided alias. But the function of the same name (`send/2`) is already auto-imported to each module from the `Kernel` module. To avoid name clashing, you need to add `import Kernel, except: [send: 2]` at the top of your module. This will autoimport the entire `Kernel` module except `send/2`. You can still use this function (and you'll need it from `send/2`) via a full qualifier: `Kernel.send/2`. The code is as follows:

```
defmodule Todo.ProcessRegistry do
  import Kernel, except: [send: 2]          <- Disables the import of Kernel.send/2

  def send(key, message) do
    case whereis_name(key) do
      :undefined -> {:badarg, {key, message}}
      pid ->
        Kernel.send(pid, message)            <- Uses Kernel.send/2
        pid
    end
  end

  ...
end
```

NOTE This implementation of the process registry is only for demonstration purposes. In real life, you don't need to write this yourself; a popular third-party library called `gproc` (https://github.com/uwiger/gproc) does it for you. In chapter 11, when you learn how to manage application dependencies, you'll replace the custom implementation with `gproc`.

9.1.3 Supervising database workers

With the process registry in place, you can begin supervising your database workers. The general approach is depicted in figure 9.4.

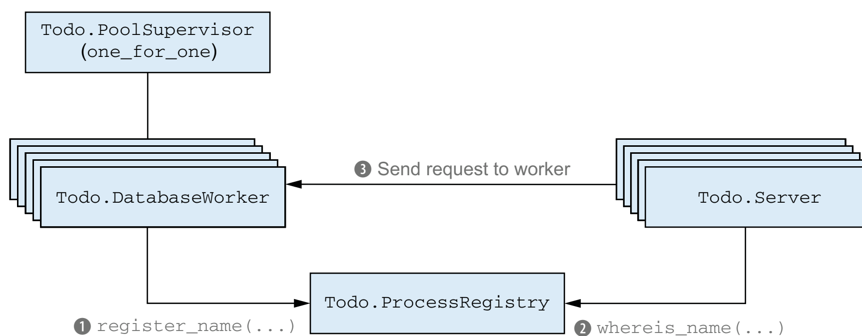

Figure 9.4 Using the process registry to discover supervised database workers

Essentially, you must do the following:

1. Have each database worker register with the process registry under a unique alias.
2. Rely on the process registry to discover proper workers.
3. Create another supervisor that starts and supervises a pool of workers.

It's worth repeating that steps 1 and 2 (process registration and discovery) are performed internally by `GenServer`, because you'll use via tuples.

Let's begin implementing the database workers. The code is presented in the following listing.

Listing 9.3 Registering workers (pool_supervision/lib/todo/database_worker.ex)

```
defmodule Todo.DatabaseWorker do
  use GenServer

  def start_link(db_folder, worker_id) do
    IO.puts "Starting the database worker #{worker_id}"

    GenServer.start_link(
      __MODULE__, db_folder,
      name: via_tuple(worker_id)          # Registers via the process registry
    )
  end

  ...

  defp via_tuple(worker_id) do
    {:via, Todo.ProcessRegistry, {:database_worker, worker_id}}     # Creates a proper via tuple
  end
end
```

This code introduces the notion of a `worker_id`, which is an integer in the range `1..pool_size`. You accept this ID in `start_link/2` and then make a full via tuple. The alias of your worker is in the form `{:database_worker, worker_id}`, which makes it possible to distinguish database workers from other types of processes that may be registered in the process registry.

You also need to modify worker interface functions. These need worker IDs instead of pids. You'll again rely on via tuples to identify the proper process.

Listing 9.4 Worker functions (pool_supervision/lib/todo/database_worker.ex)

```
defmodule Todo.DatabaseWorker do
  ...

  def store(worker_id, key, data) do
    GenServer.cast(via_tuple(worker_id), {:store, key, data})
  end

  def get(worker_id, key) do
    GenServer.call(via_tuple(worker_id), {:get, key})
  end

  ...
end
```

This is a simple modification. You send via tuples instead of pids, thus relying on your own process registry to discover the process.

It's worth noting that corresponding process-registry functions (`whereis_name`) are called from the caller process (the one calling the `store` and `get` functions). In contrast, when calling `GenServer.start_link`, `Todo.ProcessRegistry.register_name` is invoked from the process that is being registered (in this case, a database worker).

Now you can create a new supervisor that will start the pool of workers. Why are you introducing a separate supervisor? Theoretically, you could place workers under `Todo.Supervisor`, and this would work fine. But remember from the previous chapter what happens if restarts happen too often: the supervisor gives up at some point and terminates all of its children. If you keep too many children under the same supervisor, a failed restart of one child will terminate the entire system.

In this case, I made an arbitrary decision to place a distinct part of the system (the database) under a single supervisor. This approach may limit the impact of a failed restart to database operations. If restarting one database worker fails, the supervisor will terminate, which means the parent supervisor will try to restart the entire database system without touching other processes in the system. The code is presented in the next listing.

Listing 9.5 Pool supervisor (pool_supervision/lib/todo/pool_supervisor.ex)

```
defmodule Todo.PoolSupervisor do
  use Supervisor

  def start_link(db_folder, pool_size) do
    Supervisor.start_link(__MODULE__, {db_folder, pool_size})
  end

  def init({db_folder, pool_size}) do
    processes = for worker_id <- 1..pool_size do
      worker(
        Todo.DatabaseWorker, [db_folder, worker_id],
        id: {:database_worker, worker_id}
      )
    end

    supervise(processes, strategy: :one_for_one)
  end
end
```

Creates the list of child specifications

Supervisor child ID

This code is mostly straightforward. You create a list of child specifications and include it in a supervisor specification. Again, you use a `one_for_one` supervisor to ensure that each child is restarted individually.

Notice how you set up each worker to be started with the proper arguments. When the supervisor starts (or restarts) each worker, it calls `Todo.DatabaseWorker.start_link(db_folder, worker_id)`, passing the corresponding values. This ensures that each worker is properly started and registered.

Take special note of `id: {:database_worker, worker_id}`. This has no relation to the process-registry aliases you just used. In this context, you're creating a supervisor

child ID that's used internally by the supervisor to distinguish each child. By default, if you don't provide a child ID, a worker module alias (in this case, `Todo.DatabaseWorker`) is used. But here you have multiple workers handled by the same module, so you must manually define a unique ID for each worker. Otherwise, if you try to specify multiple children with the same ID, an error will be raised during supervisor startup. In this case you use a simple tuple; its second element is the internal worker ID. This ensures that each worker has a unique supervisor child ID.

9.1.4 Removing the database process

With these changes in place, the database server doesn't need to know about the worker pid. Instead, it can pass the `worker_id`, which is converted to the corresponding pid via the process registry.

This approach has an important consequence: the database server doesn't need to be a process! Because the database no longer needs to maintain workers pids (this is now part of the process-registry state), there is no state to be maintained in the database server process, so you don't need the process anymore.

You'll keep the `Todo.Database` module, though. It's an interface point for clients that allows you to contain most changes in a single module, leaving the code of the clients (to-do servers) untouched.

Let's adapt the `Todo.Database` module to rely on the new process registry. First, you'll change its `start_link` function to start the pool supervisor, as demonstrated next.

Listing 9.6 Starting the pool (pool_supervision/lib/todo/database.ex)

```
defmodule Todo.Database do
  @pool_size 3                          <-- Pool size

  def start_link(db_folder) do
    Todo.PoolSupervisor.start_link(db_folder, @pool_size)   <-- Starts the pool
  end

  ...
end
```

The `start_link/1` function delegates to the `Todo.PoolSupervisor` you just created. The knowledge of the pool size is defined using the module attribute `@pool_size`, and this is the only place in the code where you'll need to change it, if needed.

Finally, you can rewrite the implementation of database requests. The code is given in the following listing.

Listing 9.7 Database requests (pool_supervision/lib/todo/database.ex)

```
defmodule Todo.Database do
  ...

  def store(key, data) do
    key
    |> choose_worker
```

```
    |> Todo.DatabaseWorker.store(key, data)
  end

  def get(key) do
    key
    |> choose_worker
    |> Todo.DatabaseWorker.get(key)
  end

  defp choose_worker(key) do
    :erlang.phash2(key, @pool_size) + 1      <-- Chooses a worker ID
  end
end
```

Here you adapt the `store/2` and `get/1` functions to rely on the new infrastructure. Both work in a similar fashion. First you choose a worker based on the input `key`, and then you delegate to `Todo.DatabaseWorker`.

The `choose_worker/1` function takes the key, makes a numerical hash of it, normalizes it to fall in the range `0..2`, and increments by one, so that the final result is in the range `1..3`. This technique ensures affinity—requests with the same key will end up in the same worker and thus be synchronized.

This approach seems to be different than the one depicted earlier, in figure 9.4. But keep in mind that the `store` and `get` functions are called from the server processes. Because the database is no longer a process, these invocations are delegated to `Todo.DatabaseWorker` interface functions that run in the caller process (the to-do server). Consequently, the process relationship depicted in figure 9.4 is correct. Process discovery is done from the to-do servers. The corresponding code resides in the `Todo.Database` and `Todo.DatabaseWorker` modules because you used those modules to wrap the code and make clients oblivious to message-passing details.

9.1.5 Starting the system

You're almost finished, but you need to make two changes in `Todo.Supervisor`. First, you must start the process registry. Additionally, you must take into account the fact that calling `Todo.Database.start_link` now starts a supervisor process. The changes are simple:

```
defmodule Todo.Supervisor do
  ...

  def init(_) do
    processes = [
      worker(Todo.ProcessRegistry, []),                 <-- Starts the process registry
      supervisor(Todo.Database, ["./persist/"]),        <-- The child is a supervisor.
      worker(Todo.Cache, [])
    ]
    supervise(processes, strategy: :one_for_one)
  end
end
```

Take particular note how you use the `supervisor` helper function when specifying the `Todo.Database` child. By doing this, you inform the behaviour that this particular child is a supervisor. This information is mostly needed for hot code upgrades.

Keep in mind that processes are started synchronously, in the order you specify. Thus, the order in the child specification matters and isn't chosen arbitrarily. A child process always must be specified after its dependencies. In this case, you must start the registry first, because database workers depend on it.

It's worth noting up front that this isn't proper supervision of the process registry. If a registry crashes, the restarted process won't be aware of existing registered processes. You'll deal with this problem later, in section 9.2.2.

But for now, let's see how the system works. Start everything:

```
iex(1)> Todo.Supervisor.start_link
Starting process registry
Starting database worker 1
Starting database worker 2
Starting database worker 3
Starting to-do cache.
```

Now, let's verify that you can restart individual workers correctly. Because you know worker aliases are used in the process registry, you can obtain a worker's pid and terminate it:

```
iex(2)> Todo.ProcessRegistry.whereis_name({:database_worker, 2}) |>
          Process.exit(:kill)
Starting database worker 2
```

The worker is restarted, as expected, and the rest of the system is undisturbed.

9.1.6 Organizing the supervision tree

Let's stop for a moment and reflect on what you've done so far. The relationship between processes is presented in figure 9.5.

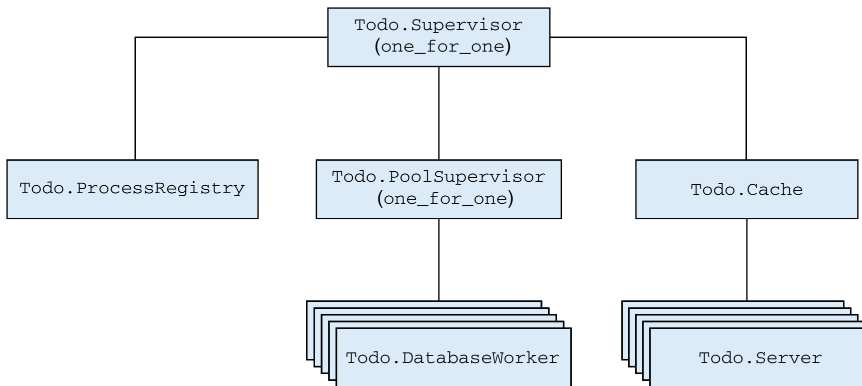

Figure 9.5 Supervision tree

This is an example of a simple *supervision tree*—a nested structure of supervisors and workers. In this case, you have a top-level to-do supervisor that is responsible for starting the entire system. Underneath, it runs the process registry, the pool supervisor, and the to-do cache.

Looking at this diagram, you can reason about how errors are handled and propagated throughout the system. If a database worker crashes, the pool supervisor will restart it, leaving the rest of the system alone. If that doesn't help, you'll exceed the maximum restart frequency, and the pool supervisor will terminate all database workers and then itself.

This will be noticed by the to-do supervisor, which will then start a fresh database pool in hopes of solving the problem. What does all this restarting get you? By restarting an entire group of workers, you effectively terminate all pending database operations and begin clean. If that doesn't help, then there's nothing more you can do, so you propagate the error up the tree (in this case, killing everything). This is the point of supervision trees—you try to recover from an error locally, affecting as few processes as possible. If that doesn't work, you move up and try to restart the wider part of the system.

OTP-compliant processes

All processes that are started directly from a supervisor should be *OTP-compliant processes*, meaning they must comply with OTP design principles and handle OTP-specific messages. OTP behaviours, such as `gen_server` and `supervisor`, and Elixir abstractions, such as `Agent` and `Task`, can be used to run OTP-compliant processes. Plain processes started by `spawn_link` aren't OTP compliant, so such processes shouldn't be started directly from a supervisor. You can freely start plain processes from workers such as `gen_servers`, but more often than not it's better to use OTP-compliant processes wherever possible. Doing so will improve the logging of your system, because crashes that take place in OTP-compliant processes are logged with more details.

Shutting down processes

An important benefit of supervision trees is the ability to stop the entire system without leaving dangling processes. When you terminate a supervisor, all of its immediate children are also terminated. If all other processes are directly or indirectly linked to those children, then they will eventually be terminated as well. Consequently, you can stop everything by terminating the top-level supervisor process.

Most often, a supervisor subtree is terminated in a controlled manner. A supervisor process will instruct its children to terminate gracefully, thus giving them the chance to do final cleanup. If some of those children are themselves supervisors, they will take down their own tree in the same way. Graceful termination of a `gen_server` worker involves invoking the `terminate/2` callback, but only if the worker process is trapping exits. Therefore, if you want to do some cleanup from a `gen_server` process, make sure you set up an exit trap from `init/1` callback.

Because graceful termination involves the possible execution of cleanup code, it may take longer than desired. An option called a *shutdown strategy* lets you control how

long the supervisor will give its child to terminate gracefully. The default strategy is to wait at most five seconds for the child to terminate. If the child doesn't terminate in this time, it will be forcefully terminated. You can change this behavior by specifying `shutdown: shutdown_strategy` for each supervisor child, passing an integer indicating a time in milliseconds that's allowed for the child to terminate gracefully. Alternatively, you can pass the atom `:infinity`, which instructs the supervisor to wait indefinitely for the child to terminate. Finally, you can pass the atom `:brutal_kill`, telling the supervisor to immediately terminate the child in a forceful way. The forceful termination is done by sending a `:kill` exit signal to the process, similarly to what you did with `Process.exit(pid, :kill)`.

AVOIDING PROCESS RESTARTING

By default, a supervisor restarts a terminated process, regardless of the exit reason. Even if the process terminates with the reason `:normal`, it will be restarted. Sometimes you may want to alter this behavior.

For example, consider a process that handles an HTTP request or a TCP connection. If such a process fails, the socket will be closed, and there's no point in restarting the process (the remote party will be disconnected anyway). Regardless, you want to have such processes under a supervision tree, because this makes it possible to terminate the entire supervisor subtree without leaving dangling processes. In this situation, you can set up a *temporary* worker by using `worker(module, args, restart: :temporary)` in the supervisor specification. A temporary worker isn't restarted on termination.

Another option is a *transient* worker, which is restarted only if it terminates abnormally. Transient workers can be used for processes that may terminate normally, as part of the standard system workflow. For example, in the caching scheme, you use to-do server processes to keep to-do lists in memory. You may want to terminate individual servers normally if they haven't been used for a while. But if a server crashes abnormally, you want to restart it. This is exactly how transient workers function. A transient worker can be specified with `worker(module, args, restart: :transient)`.

This concludes our initial take on fine-grained supervision. You've made a number of changes that minimize the effects of errors. There's still a lot of room for improvements, and you'll continue extending the system in the next section, where you'll learn how to start workers dynamically.

9.2 Starting workers dynamically

The impact of a database-worker error is now confined to a single worker. It's time to do the same thing for to-do servers—the processes that manage to-do lists in memory. You'll use roughly the same approach as you did with database workers: you'll run each to-do server under a supervisor and register the process in the process registry.

There is a twist, though. Unlike database workers, to-do servers are created dynamically, when needed. Initially, no to-do server is running; each is created on demand when you call `Todo.Cache.server_process/1`. This effectively means you can't specify

supervisor children up front—you don't know how many children you need or what arguments should be passed when starting child processes.

For such cases, you need a dynamic supervisor that can start a child on demand. In OTP parlance, such supervisors have a slightly misleading name: `simple_one_for_one` supervisors.

9.2.1 simple_one_for_one supervisors

This is a special case of the `one_for_one` strategy, with some different properties:

- There can be multiple children, but they're all started by the same function. Hence, the child specification must contain exactly one element.
- No child is started upfront. Children can be started dynamically, on demand, by calling `Supervisor.start_child/2`.

NOTE It's possible to use other restart strategies to start children dynamically. You can, for example, use the `one_for_one` strategy and still start your children via `Supervisor.start_child/2`. But if all children are of the same type, it's more idiomatic to use `simple_one_for_one`, to indicate that you're starting children dynamically and that all children will be of the same type.

Let's create a to-do server supervisor. The implementation is provided in the following listing.

Listing 9.8 To-do server supervisor (dynamic_workers/lib/todo/server_supervisor.ex)

```
defmodule Todo.ServerSupervisor do
  use Supervisor

  def start_link do
    Supervisor.start_link(__MODULE__, nil,
      name: :todo_server_supervisor
    )
  end

  def start_child(todo_list_name) do
    Supervisor.start_child(
      :todo_server_supervisor,
      [todo_list_name]
    )
  end

  def init(_) do
    supervise(
      [worker(Todo.Server, [])],
      strategy: :simple_one_for_one
    )
  end
end
```

Annotations: Local registration of the supervisor (`name: :todo_server_supervisor`); References the supervisor via a local alias (`:todo_server_supervisor,`); Arguments passed to Todo.Server.start_link (`[todo_list_name]`); Child specification (`[worker(Todo.Server, [])],`)

This resembles other supervisors you've seen, with a couple of twists. First, you make the supervisor register locally under an alias, `:todo_server_supervisor`. This is

needed for easier starting of children. When you want to start a child, you need to request this operation from the supervisor process. Remember, to issue a request to a specific process, you must either have its pid or know its registered alias (if it has one). To make things simple, you use an alias here.

In the `init/1` function, you still have to provide a child specification. Consider it a template that's used when starting a child. The specification of a dynamic child consists of the responsible module (`Todo.Server`) and the list of predefined, constant arguments (`[]`) that will be passed to `start_link` when starting the child.

Finally, the `start_child/1` function delegates to `Supervisor.start_child/2`. Notice how you use the `:todo_server_supervisor` alias to issue a request to the proper supervisor process.

When starting the child, you provide the list of additional arguments for `Todo.Server.start_link`. This list is appended to the one given in the child specification in `init/1`, to form the complete list of arguments passed to `Todo.Server.start_link`. The final call used to start a child thus translates to `Todo.Server.start_link(todo_list_name)`.

With this in place, you have to make the to-do server register itself to the process registry during startup. In addition, you'll implement a `whereis/1` function that returns the pid of the to-do server:

```
defmodule Todo.Server do
  use GenServer

  def start_link(name) do
    IO.puts "Starting to-do server for #{name}"
    GenServer.start_link(Todo.Server, name, name: via_tuple(name))    # Registers in the process registry
  end

  defp via_tuple(name) do
    {:via, Todo.ProcessRegistry, {:todo_server, name}}
  end


  def whereis(name) do
    Todo.ProcessRegistry.whereis_name({:todo_server, name})    # Discovers the to-do server
  end

  ...
end
```

With this scheme set up, you don't need to keep a mapping of to-do list names to process IDs in the to-do cache process. This mapping is now maintained in the process registry, and you can invoke `Todo.Server.whereis/1` to discover the process.

But you still need the cache process itself to prevent a special case of a race condition. Recall that when clients want to manipulate a particular to-do list, they use the `Todo.Cache.server_process/1` function to get (or create) the corresponding to-do server process. If two or more concurrent clients try to call this function at the same time for the same nonexistent list, you should create the corresponding server process

only once. Therefore, you'll make a slight modification in the to-do cache implementation. In particular, `server_process/1` needs to have the following flow:

1. In the client process, check whether the to-do server process exists. If yes, return the corresponding pid.
2. Otherwise, issue a call into the to-do cache process.
3. In the to-do cache process, recheck once again if the desired process exists. If yes, return the corresponding pid.
4. Otherwise, in the to-do cache process, create the process (via `Todo.ServerSupervisor`) and return the corresponding pid.

This ensures that multiple requests for the same nonexistent process will result in the creation of only one process, while other requests will wait for that process to be created. But processes that request an existing to-do list server won't even enter the cache process. For the sake of brevity, I omit the code, which can be found in dynamic_workers/lib/todo/cache.ex.

The final change must be made in the `Todo.Supervisor` specification, where you additionally start the new `Todo.ServerSupervisor`, as illustrated in the next listing.

Listing 9.9 `Todo.Supervisor` (dynamic_workers/lib/todo/supervisor.ex)

```
defmodule Todo.Supervisor do
  ...

  def init(_) do
    processes = [
      worker(Todo.ProcessRegistry, []),
      supervisor(Todo.Database, ["./persist/"]),
      supervisor(Todo.ServerSupervisor, []),      <-- New supervisor
      worker(Todo.Cache, [])
    ]
    supervise(processes, strategy: :one_for_one)
  end
end
```

With this in place, you can check if everything works. Start the shell and the entire system:

```
iex(1)> Todo.Supervisor.start_link
Starting process registry
Starting database worker 1
Starting database worker 2
Starting database worker 3
Starting to-do cache
```

Now, let's get one to-do server:

```
iex(2)> bobs_list = Todo.Cache.server_process("Bob's list")
Starting to-do server for Bob's list
#PID<0.65.0>
```

Repeating the request doesn't start another server:

```
iex(3)> bobs_list = Todo.Cache.server_process("Bob's list")
#PID<0.65.0>
```

In contrast, using a different to-do list name creates another process:

```
iex(4)> alices_list = Todo.Cache.server_process("Alice's list")
Starting to-do server for Alice's list
#PID<0.68.0>
```

Crash one to-do server:

```
iex(5)> Process.exit(bobs_list, :kill)
Starting to-do server for Bob's list
```

You can immediately see that Bob's to-do server has been restarted. The subsequent call to `Todo.Cache.server_process/1` will return a different pid:

```
iex(6)> Todo.Cache.server_process("Bob's list")
#PID<0.70.0>
```

Of course, Alice's server remains undisturbed:

```
iex(7)> Todo.Cache.server_process("Alice's list")
#PID<0.68.0>
```

The supervision tree of the new code is presented in figure 9.6. The diagram clearly depicts how you supervise each process, limiting the effect of unexpected errors.

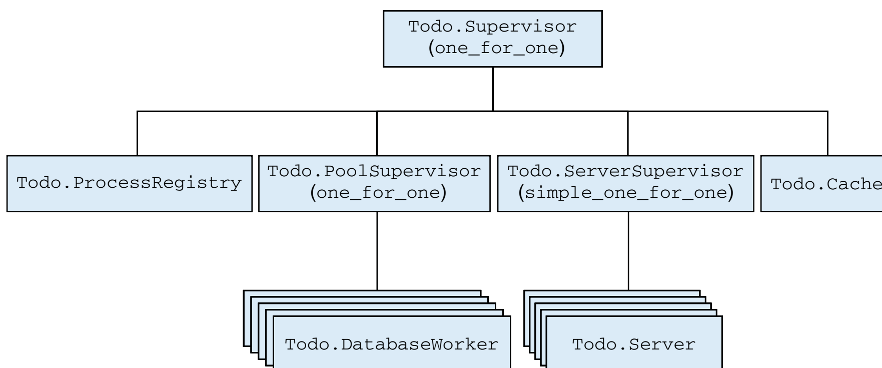

Figure 9.6 Supervising to-do servers

9.2.2 Restart strategies

So far, you've been using only `one_for_one` and `simple_one_for_one` strategies. They both handle an error by starting a new process in place of a crashed child, leaving other children alone. Occasionally, you may need a different behavior. The supervisor provides two additional restart strategies:

- `one_for_all`—When a child crashes, the supervisor terminates all other children as well and then starts all children.
- `rest_for_one`—When a child crashes, the supervisor terminates all processes from the child specification that are listed after the crashed child. Then the supervisor starts new child processes in place of the terminated ones.

Let's see this in practice. Your error handling and recovery still aren't quite correct: if a process registry is restarted, the alias-to-pid mapping is lost. Remember, the new process starts with a clean state. In this case, this is a big problem, because you can't reach existing processes via the registry.

What can you do about this? First, you must recognize that the process registry is very important for the entire system. Without it, the rest of the system can't function. Although you can afford to lose a to-do server occasionally, losing the process registry renders everything unusable. Consequently, you should keep the process-registry code as simple as possible, thus minimizing possible errors. Here, the process registry just stores data to a hash dict and reads from it. Assuming you trust the `HashDict` implementation to be correct, you can safely say that the code is error free and the chance of it crashing is minimal.

Still, you should have a backup plan—a clean way to handle the case of the process registry crashing. Given that nothing else can function without the registry, your best bet is to kill everything else if the registry terminates. To do this, you need a new, top-level supervisor that will start and supervise the process registry and the rest of the to-do system. When the registry crashes, you want to terminate the rest of the system.

This is exactly the use case for the `one_for_all` and `rest_for_one` strategies, both of which terminate related processes when one process crashes. But which strategy is more appropriate for this situation? Remember, you must always ask yourself: if one process crashes, what should happen to the remaining ones? Here, when the process registry crashes, you want to kill everything else. But if the rest of the to-do system crashes, you should leave the process registry alone. This is the correct approach because the process registry doesn't depend on the rest of the system and should therefore continue to work properly even when everything else crashes.

Given the line of thinking just described, your choice should be the `rest_for_one` restart strategy. The corresponding supervision tree is shown in figure 9.7.

Using the `rest_for_one` strategy means a crash of the process registry will take down the to-do system as well. But when the to-do system crashes, the process registry will remain running, which is fine. Remember that you set up monitors in the process registry; when registered processes terminate, you'll unregister them, and the state of the registry will be consistent.

As an exercise, you can try to implement this supervision tree yourself. Doing so should be fairly simple, but if you get stuck, the solution can be found in the proper_registry_supervision folder.

With this, you're finished making your to-do system fault tolerant. You've introduced additional supervisor processes to the system, and you've also managed to

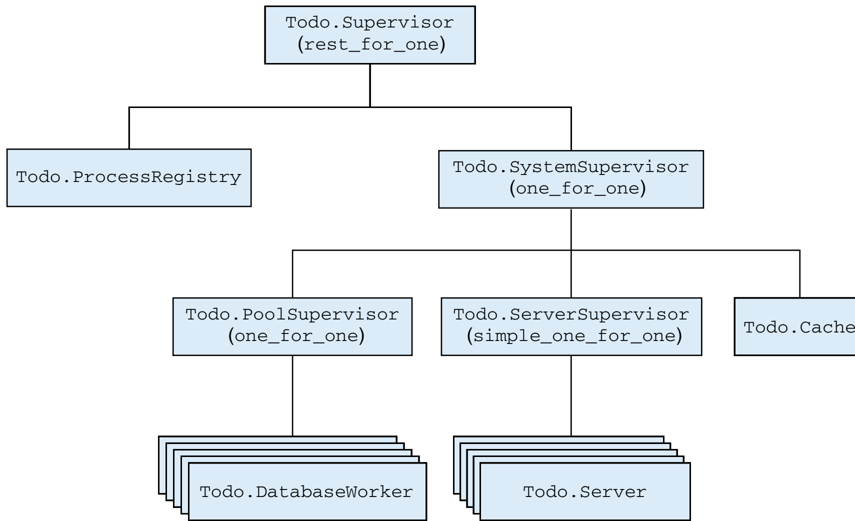

Figure 9.7 `rest_for_one` **supervision**

simplify some other parts (removing the to-do cache and database server processes). You'll make many more changes to this system, but for now let's leave it and look at some important practical considerations.

9.3 "Let it crash"

In general, when you develop complex systems, you should employ supervisors to do your error handling and recovery. With properly designed supervision trees, you can limit the impact of unexpected errors, and the system will hopefully recover. I can personally testify that supervisors have helped me in occasional weird situations in production, keeping the running system stable and saving me from unwanted phone calls in the middle of the night. It's also worth noting that OTP provides logging facilities, so process crashes are logged and you can see that something went wrong. It's even possible to set up an event handler that will be triggered on every process crash, thus allowing you to perform custom actions—for example, sending an email or report to an external system.

An important consequence of this style of error handling is that the `main` worker code is liberated from paranoid, defensive `try/catch` constructs. Usually these aren't needed, because you use supervisors to handle error recovery. Joe Armstrong, one of the inventors of Erlang, described such a style in his Ph.D thesis (www.erlang.org/download/armstrong_thesis_2003.pdf) as *intentional programming*. Using this approach, the code states the programmer's intention, rather than being cluttered with all sorts of defensive constructs.

This style is also known as *let it crash*. In addition to making the code shorter and more focused, *let it crash* promotes a clean-slate recovery. Remember, when a new process starts, it starts with a new state, which should be consistent. Furthermore, the

message queue (mailbox) of the old process is thrown away. This will cause some requests in the system to fail. But the new process starts fresh, which gives it a better chance to resume normal operation.

Let it crash can initially seem confusing, and people may mistake it for the *let everything crash* approach. There are two important situations in which you should explicitly handle an error:

- Critical processes that shouldn't crash
- An error that can be dealt with in a meaningful way

9.3.1 Error kernel

The first case relates to what is informally called a system's *error kernel*—processes that are critical for the entire system to work and whose state can't be restored in a simple and consistent way. Such processes are the heart of your system, and you generally don't want them to crash. In the to-do system, you could say that the process registry is your error kernel. Without the registry, other processes can't connect to each other, and the entire system stops working.

In general, you should keep the code of such important processes as simple as possible. The less logic happens in the process, the smaller the chance of a process crash. If the code of your error-kernel process is complex, consider splitting it into two processes: one that holds a state, and another that does the actual work. The former process then becomes extremely simple and thus unlikely to crash, whereas the worker process can be removed from the error kernel (because it no longer maintains a critical state).

In addition, it makes sense to include defensive `try/catch` statements in each `handle_*` callback of a critical process, thus preventing a process from crashing. Here's a simple sketch of the idea:

```
def handle_call(message, _, state) do
  try
    new_state =
      state
      |> transformation_1
      |> transformation_2
      ...

    {:reply, response, new_state}
  catch _, _ ->
    {:reply, {:error, reason}, state}      # Catches all errors, and uses the original state
  end
end
```

This snippet illustrates how immutable data structures allow you to implement a fault-tolerant server. While processing a request, you make a series of transformations on the state. If anything bad happens, you use the initial state, effectively performing a rollback of all changes. This preserves state consistency while keeping the process constantly alive.

Keep in mind that this technique doesn't completely guard against a process crash. For example, you can always kill a process by invoking `Process.exit(pid, :kill)`, because a `:kill` exit reason can't be intercepted even if you're trapping exits. Therefore, you should always have a recovery plan for a crash of a critical process. This plan amounts to setting up a proper supervision hierarchy to ensure termination of all dependent processes in case of an error-kernel process crash.

9.3.2 Handling expected errors

The whole point of *let it crash* is to leave recovery of unexpected errors to supervisors. But if you can predict an error and you have a way to deal with it, there's no reason to let the process crash. Here's a simple example. Look at the `:get` request in the database worker:

```
def handle_call({:get, key}, _, db_folder) do
  data = case File.read(file_name(db_folder, key)) do
    {:ok, contents} -> :erlang.binary_to_term(contents)
    _ -> nil                                          # Handles a file-read error
  end

  {:reply, data, db_folder}
end
```

When handling a `get` request, you try to read from the file, covering the case when this reading fails. If you don't succeed, you return `nil`, treating this case as if an entry for the given key isn't in the database.

You can do better. Consider using an error only when a file isn't available. This error is identified with `{:error, :enoent}`, so the corresponding code would look like this:

```
case File.read(...) do
  {:ok, contents} -> do_something_with(contents)
  {:error, :enoent} -> nil
end
```

Notice how you rely on pattern matching here. If neither of these two expected situations happens, a pattern match will fail, and so will your process. This is the idea of *let it crash*. You deal with expected situations (the file is either available or doesn't exist), ignoring anything else that can go wrong (for example, you don't have permissions). Personally, I don't even regard this as error handling. It's a normal execution path—an expected situation that can and should be dealt with. It's certainly not something you should let crash.

In contrast, when storing data, you use `File.write!/2` (notice the exclamation), which may throw an exception and crash the process. If you didn't succeed in saving the data, then your database worker has failed, and there's no point in hiding this fact. Better to fail fast, which will cause an error that will be logged and (hopefully) noticed and fixed.

Of course, restarting may or may not help. For example, if a bug in the system allows multiple workers to write to the same file, then restarting will fix this situation—

you let one process crash, and after restart, things work again. Occasional requests will fail, but the system as a whole will provide service. In a different case, perhaps you don't have proper file permissions; then restarting won't help. But the supervisor will give up and crash itself, and the system will quickly come to a halt, which is probably a good thing. No point in working if you can't persist the data.

As a general rule, if you know what to do with an error, you should definitely handle it. Otherwise, for anything unexpected, let the process crash, and ensure proper error isolation and recovery via supervisors.

9.3.3 Preserving the state

Keep in mind that state isn't preserved when a process is restarted. Remember from chapter 5 that a process state is its own private affair. Thus, when a process crashes, the memory it occupied is reclaimed, and the new process starts with the new state. Thus it has the important advantage of starting clean. Perhaps a process crashed due to an inconsistent state, and starting fresh may fix an error.

That said, in some cases you want the process state to survive the crash. This isn't something provided out of the box; you need to implement it yourself. The general approach is to save the state out of the process (for example, in another process or to a file) and then restore the state when the successor process is started.

You already have this functionality in the to-do server. Recall that you have a simple database system that persists to-do lists to the disk. When the to-do server is started, the first thing it tries to do is restore the data from the database. This makes it possible for the new process to inherit the state of the old one.

In general, be careful when preserving state. As you learned in chapter 4, a typical change in a functional data abstraction goes through chained transformations:

```
new_state =
  state
  |> transformation_1(...)
  ...
  |> transformation_n(...)
```

As a rule, the state should be persisted after all transformations are completed. Only then can you be certain that your state is consistent, so this is a good opportunity to save it. For example, you do this in the to-do cache after you modify the internal data abstraction:

```
def handle_cast({:add_entry, new_entry}, {name, todo_list}) do
  new_state = TodoList.add_entry(todo_list, new_entry)
  Todo.Database.store(name, new_state)              <-- Persists the state
  {:noreply, {name, new_state}}
end
```

TIP Persistent state can have a negative effect on restarts. Let's say an error is caused by a state that is somehow invalid (perhaps due to a bug). If this state is persisted, your process can never restart successfully, because the process will restore the invalid state and then crash again (either on start or when

handling a request). Thus you should be careful when persisting state. If you can afford to, it's better to start clean and terminate all other dependent processes, just as you did with the process registry.

9.4 Summary

This concludes the topic of fine-grained supervision. You've covered a lot of ground and made your system more resilient to errors. Before continuing, here are some points worth repeating:

- Supervisors allow you to localize the impact of an error, keeping unrelated parts of the system undisturbed.
- Each process should reside somewhere in a supervision tree. This makes it possible to terminate the entire system (or an arbitrary sub-part of it) by terminating the supervisor.
- When a process crashes, its state is lost. You can deal with it by storing state outside the process, but more often than not, it's best to start with a clean state.
- In general, you should handle unexpected errors through a proper supervision hierarchy. Explicit handling through a `try` construct should be used only when you have a meaningful way to deal with an error.

After this detailed treatment of supervision trees, it's time to move on to the next topic: shared state.